The Nineteenth Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

Problem A

Jill's Bike

Input File: bike.in

Jill Bates hates climbing hills. Jill rides a bicycle everywhere she goes, but she always wants to go the easiest and shortest way possible. The good news is that she lives in Greenhills, which has all its roads laid out in a strictly rectangular grid—east-west roads are *streets*; north-south roads are *avenues* and the distance between any two adjacent grid points is the same. The bad news is that Greenhills is very hilly and has many one-way roads.

In choosing a route between where she starts and where she ends, Jill has three rules:

1. Avoid any climb of more than 10 meters between adjacent grid points.
2. Never go the wrong way on a one-way road.
3. Always travel the shortest possible route.

Your program should help Jill find an acceptable route.

Input

The input file contains data in the following form:

- The first line contains two integers, separated by one or more spaces. The first integer *n* represents the number of streets, and the second integer *m* represents the number of avenues, 1_ *n* _20, 1_ *m* _20.
- The next *n* lines contain the altitudes of grid points. Each line represents a street and contains a sequence of *m* integers separated by one or more spaces. These integers represent the altitude in meters of the grid points along that street. Even if a particular street and avenue have no intersection, the altitude is still given for that grid point.
- One or more lines follow that define the one-way roads. Each road is represented by two pairs of integers, separated by one or more spaces, in the form:

  street avenue street avenue

  The first street and avenue define the starting point of the road and the second pair define the ending point. Since Greenhills is a strict grid, if the two points are not adjacent in the grid, the road passes through all the intervening grid points. For example,

  ```
  5 7 5 10
  ```

  represents roads 5-7 to 5-8, 5-8 to 5-9, and 5-9 to 5-10. Road definitions are terminated by a line containing four zeroes in the above format.
- Finally, one or more lines will follow that contain pairs of grid points between which Jill wants to find an optimal path, in the form:

  street avenue street avenue

  As before, the integer pairs are separated by one or more spaces. The end of the input set is defined by a line containing four zeroes, formatted as before.

You may assume that all street and avenue numbers are within the bounds defined by the first line of input, and that all road definitions are strictly north-south or east-west.

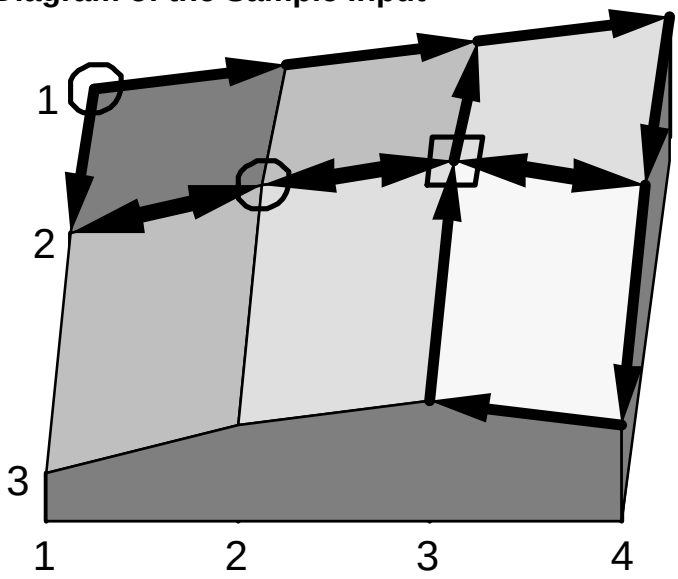
Output

For each path query in the input file, output a sequence of grid points, from the starting grid point to the ending grid point, which meets Jill's three rules. Output grid points as '*street*`-`*avenue*' separated by the word '`to`'. If there is more than one path that meets Jill's criteria, any such path will be acceptable. If no route satisfies all the criteria, or if the starting and ending grid points are the same, output an appropriate message to that effect. Output a blank line between each output set.

Sample Input

```
3 4
10 15 20 25
19 30 35 30
10 19 26 20
1 1 1 4
2 1 2 4
3 4 3 3
3 3 1 3
1 4 3 4
2 4 2 1
1 1 2 1
0 0 0 0
1 1 2 2
2 3 2 3
2 2 1 1
0 0 0 0
```

Diagram of the Sample Input



Output for the Sample Input

```
1-1 to 1-2 to 1-3 to 1-4 to 2-4 to 2-3 to 2-2

To get from 2-3 to 2-3, stay put!

There is no acceptable route from 2-2 to 1-1.
```

The Nineteenth Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

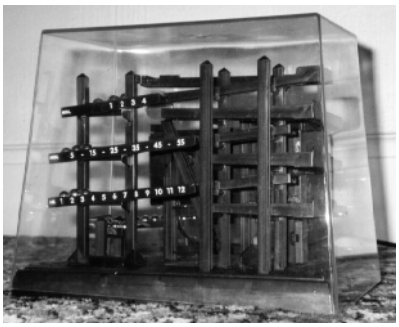
Problem B

Tempus et mobilius

Time and motion

Input File: clock.in

Tempus est mensura motus rerum mobilium.
Time is the measure of movement.
—Auctoritates Aristotelis



…and movement has long been used to measure time. For example, the ball clock is a simple device which keeps track of the passing minutes by moving ball-bearings. Each minute, a rotating arm removes a ball bearing from the queue at the bottom, raises it to the top of the clock and deposits it on a track leading to indicators displaying minutes, five-minutes and hours. These indicators display the time between 1:00 and 12:59, but without 'a.m.' or 'p.m.' indicators. Thus 2 balls in the minute indicator, 6 balls in the five-minute indicator and 5 balls in the hour indicator displays the time 5:32.

Unfortunately, most commercially available ball clocks do not incorporate a date indication, although this would be simple to do with the addition of further carry and indicator tracks. However, all is not lost! As the balls migrate through the mechanism of the clock, they change their relative ordering in a predictable way. Careful study of these orderings will therefore yield the time elapsed since the clock had some specific ordering. The length of time which can be measured is limited because the orderings of the balls eventually begin to repeat. Your program must compute the time before repetition, which varies according to the total number of balls present.

Operation of the Ball Clock

Every minute, the least recently used ball is removed from the queue of balls at the bottom of the clock, elevated, then deposited on the minute indicator track, which is able to hold four balls. When a fifth ball rolls on to the minute indicator track, its weight causes the track to tilt. The four balls already on the track run back down to join the queue of balls waiting at the bottom in reverse order of their original addition to the minutes track. The fifth ball, which caused the tilt, rolls on down to the five-minute indicator track. This track holds eleven balls. The twelfth ball carried over from the minutes causes the five-minute track to tilt, returning the eleven balls to the queue, again in reverse order of their addition. The twelfth ball rolls down to the hour indicator. The hour indicator also holds eleven balls, but has one extra fixed ball which is always present so that counting the balls in the hour indicator will yield an hour in the range one to twelve. The twelfth ball carried over from the five-minute indicator causes the hour indicator to tilt, returning the eleven free balls to the queue, in reverse order, before the twelfth ball itself also returns to the queue.

Input

The input defines a succession of ball clocks. Each clock operates as described above. The clocks differ only in the number of balls present in the queue at one o'clock when all the clocks start. This number is given for each clock,

one per line and does not include the fixed ball on the hours indicator. Valid numbers are in the range 27 to 127. A zero signifies the end of input.

Output

For each clock described in the input, your program should report the number of balls given in the input and the number of days (24-hour periods) which elapse before the clock returns to its initial ordering.

Sample Input

```
30
45
0
```

Output for the Sample Input

```
30 balls cycle after 15 days.
45 balls cycle after 378 days.
```

The Nineteenth Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

Problem C

Variable Radix Huffman Encoding

Input File: coding.in

Huffman encoding is a method of developing an optimal encoding of the symbols in a *source alphabet* using symbols from a *target alphabet* when the frequencies of each of the symbols in the source alphabet are known. Optimal means the average length of an encoded message will be minimized. In this problem you are to determine an encoding of the first N uppercase letters (the source alphabet, S_1 through S_N, with frequencies f_1 through f_N) into the first R decimal digits (the target alphabet, T_1 through T_R).

Consider determining the encoding when R=2. Encoding proceeds in several passes. In each pass the two source symbols with the lowest frequencies, say S_1 and S_2, are grouped to form a new "combination letter" whose frequency is the sum of f_1 and f_2. If there is a tie for the lowest or second lowest frequency, the letter occurring earlier in the alphabet is selected. After some number of passes only two letters remain to be combined. The letters combined in each pass are assigned one of the symbols from the target alphabet. The letter with the lower frequency is assigned the code 0, and the other letter is assigned the code 1. (If each letter in a combined group has the same frequency, then 0 is assigned to the one earliest in the alphabet. For the purpose of comparisons, the value of a "combination letter" is the value of the earliest letter in the combination.) The final code sequence for a source symbol is formed by concatenating the target alphabet symbols assigned as each combination letter using the source symbol is formed. The target symbols are concatenated in the reverse order that they are assigned so that the first symbol in the final code sequence is the last target symbol assigned to a combination letter. The two illustrations below demonstrate the process for R=2.

Symbol	Frequency
A	5
B	7
C	8
D	15

Pass 1: A and B grouped
Pass 2: {A,B} and C grouped
Pass 3: {A,B,C} and D grouped
Resulting codes: A=110, B=111, C=10, D=0
Avg. length = (3*5+3*7+2*8+1*15)/35=1.91

Symbol	Frequency
A	7
B	7
C	7
D	7

Pass 1: A and B grouped
Pass 2: C and D grouped
Pass 3: {A,B} and {C,D} grouped
Resulting codes: A=00, B=01, C=10, D=11
Avg. length = (2*7+2*7+2*7+2*7)/28=2.00

When R is larger than 2, R symbols are grouped in each pass. Since each pass effectively replaces R letters or combination letters by 1 combination letter, and the last pass must combine R letters or combination letters, the source alphabet must contain $k*(R-1)+R$ letters, for some integer k. Since N may not be this large, an appropriate number of fictitious letters with zero frequencies must be included. These fictitious letters are not to be included in the output. In making comparisons, the fictitious letters are later than any of the letters in the alphabet.

Now the basic process of determining the Huffman encoding is the same as for the R=2 case. In each pass, the R letters with the lowest frequencies are grouped, forming a new combination letter with a frequency equal to the

sum of the letters included in the group. The letters that were grouped are assigned the target alphabet symbols 0 through *R*–1. 0 is assigned to the letter in the combination with the lowest frequency, 1 to the next lowest frequency, and so forth. If several of the letters in the group have the same frequency, the one earliest in the alphabet is assigned the smaller target symbol, and so forth. The illustration below demonstrates the process for *R*=3.

Symbol	Frequency
A	5
B	7
C	8
D	15

Pass 1: ? (fictitious symbol), A and B are grouped
Pass 2: {?,A,B}, C and D are grouped
Resulting codes: A=11, B=12, C=0, D=2
Avg. length = (2*5+2*7+1*8+1*15)/35=1.34

Input

The input will contain one or more data sets, one per line. Each data set consists of an integer value for *R* (between 2 and 10), an integer value for *N* (between 2 and 26), and the integer frequencies f_1 through f_N, each of which is between 1 and 999. The end of data for the entire input is the number 0 for *R*; it is not considered to be a separate data set.

Output

For each data set, display its number (numbering is sequential starting with 1) and the average target symbol length (rounded to two decimal places) on one line. Then display the *N* letters of the source alphabet and the corresponding Huffman codes, one letter and code per line. The examples below illustrate the required output format.

Sample Input

```
2 5 5 10 20 25 40
2 5 4 2 2 1 1
3 7 20 5 8 5 12 6 9
4 6 10 23 18 25 9 12
0
```

Output for the Sample Input

```
Set 1; average length 2.10
    A: 1100
    B: 1101
    C: 111
    D: 10
    E: 0

Set 2; average length 2.20
    A: 11
    B: 00
    C: 01
    D: 100
    E: 101

Set 3; average length 1.69
    A: 1
    B: 00
    C: 20
    D: 01
    E: 22
    F: 02
    G: 21

Set 4; average length 1.32
    A: 32
    B: 1
    C: 0
    D: 2
    E: 31
    F: 33
```

The Nineteenth Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

Problem D

Sail Race

Input File: sail.in

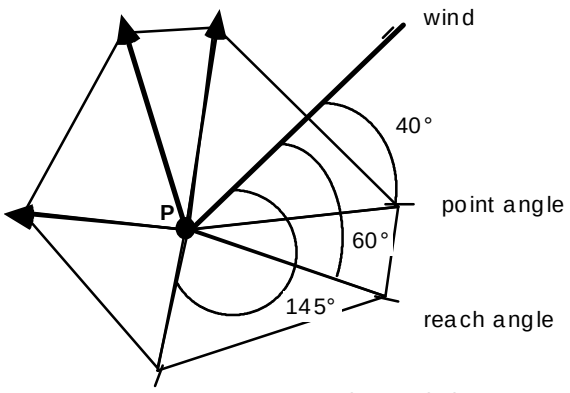
The Atlantic Coastal Mariners (ACM) sailing club is building a race planning tool to estimate durations of sailboat races with various race courses, wind directions, and types of sailboats. You must write a program to help with that task.

A race course is defined by marks with up to 10 marks per race course. A sailboat must sail to all marks in the specified order. The marks are identified as *x*- and *y*-coordinates on a hypothetical grid with a single unit equal to one nautical mile (nm). The positive *y*-axis is oriented due north and the positive *x*-axis is oriented due east. The race course is in open waters without any navigational limitations.

For purposes of this planning tool, the only driving force controlling a sailboat is the wind. The wind determines the sailboat's speed of advance and limits its direction of travel. The wind is constant for the duration of each race and is specified in terms of the direction from which the wind is blowing and its speed in nautical miles per hour (kts). Wind direction is specified as a compass bearing in degrees measured clockwise from 000.0° as north.

Sailboats cannot steer any closer to the wind than a given "point angle" off the wind direction. In order to make progress closer to the wind direction, the sailboat must tack back and forth across the wind, steering no closer to the wind than its point angle. Each time the sailboat tacks or passes a mark it incurs a tack penalty. For this simulation, each sailboat will travel each leg of a race (the portion of a race between successive marks) with the minimum number of tacks and the minimum possible distance. Courses and directions are specified as compass bearings in degrees measured clockwise from 000.0° as north.

The speed of a sailboat is determined by the sailboat design, wind speed, and direction steered relative to the wind. In the figure, the wind direction is 45° and the point angle is 40°. This means then that this sailboat cannot steer between 5° and 85° because it cannot point that closely into the wind.



Point angle is 40°
Reach angle is 60°
Downwind angle is 145°.

Sailboat cannot travel from P in this direction

Sailboat traveling from P in this direction moves at point speed ratio times wind speed

Sailboat traveling from P in this direction moves at reach speed ratio times wind speed

Sailboat traveling from P in this direction moves at downwind speed ratio times wind.

Each angle is measured as the smallest angle between the direction of travel of the sailboat and the direction from which the wind is blowing.

For this problem, the ratio of sailboat speed to wind speed is one of three ratios, selected as shown in the table below according to the angle off the wind :

Angle off wind	Applicable ratio
_ point angle and < reach angle	point speed ratio
_ reach angle and < downwind angle	reach speed ratio
_ downwind angle	downwind speed ratio

For instance, if the boat is steering at an angle off the wind which is between the reach angle and downwind angle then

$$\text{boat speed} = \text{reach speed ratio} \times \text{wind speed}$$

Input

Your solution must accept multiple input data sets. Each data set represents a different race course to be evaluated for a single sailboat. The data set begins with a line with 4 numbers: wind direction (real), wind speed (real), tack penalty (real), and number of marks *n* (integer). The next line contains six real numbers: point angle, point speed ratio, reach angle, reach speed ratio, downwind angle, downwind speed ratio.

The subsequent *n* lines of the data set represent the *n* race marks in the order in which they must be reached. Each line begins with a 2-character mark id followed by the *x*-coordinate then *y*-coordinate of the mark.

The end of input is denoted by a line of four 0's.

Output

The output for your program consists of various data calculated for each input data set. Values should be presented with the following precisions and units.

Courses, tacks, directions	0.1 degree	Distance	0.01 nm
Speed	0.1 kts	Time	0.01 hours

Output for each race begins with a header containing the number of the data set (1 for the first, 2 for the second, etc.) and the number of legs. The next line is the total length of the race course, measured as the sum of distances between successive marks.

For each leg of the course, the leg number, beginning and ending mark id's, course from the beginning to end marks of the leg, and the leg distance is presented. This is followed by a listing of the tacks necessary to complete the leg. The tacks for each race are numbered sequentially, with tack numbers beginning with 1 for each race. For

each tack, the tack number, the projected sailboat speed, the course steered, and the length of that tack are presented.

The summary output for each data set includes the total number of tacks, the total distance traveled for the race, the estimated race duration, and the total tack penalty time incurred by the sailboat after leaving the first mark.

The exact format of the output is not specified, but all output should be organized so that it is in the specified order, appropriately labeled and follows given numeric specifications.

Sample Input

```
45 10 .1 6
45 0.5 90 0.75 135 0.67
M1 15 10
M2 25 20
M3 22 30
M4 5 25
M5 10 15
M6 10 10
0 0 0 0
```

Output for the Sample Input

```
========================
Race 1 has 5 legs
The race layout is  58.48 nm long
----------------------------

Leg 1 from Mark M1 to M2 == > Direction:  45.0  Distance:  14.14 nm
Tack 1 ==> Speed:  5.0   Direction:  90.0  Distance: 10.00 nm
Tack 2 ==> Speed:  5.0   Direction:   0.0  Distance: 10.00 nm

Leg 2 from Mark M2 to M3 == > Direction: 343.3  Distance:  10.44 nm
Tack 3 ==> Speed:  5.0   Direction: 343.3  Distance: 10.44 nm

Leg 3 from Mark M3 to M4 == > Direction: 253.6  Distance:  17.72 nm
Tack 4 ==> Speed:  6.7   Direction: 253.6  Distance: 17.72 nm

Leg 4 from Mark M4 to M5 == > Direction: 153.4  Distance:  11.18 nm
Tack 5 ==> Speed:  7.5   Direction: 153.4  Distance: 11.18 nm

Leg 5 from Mark M5 to M6 == > Direction: 180.0  Distance:   5.00 nm
Tack 6 ==> Speed:  6.7   Direction: 180.0  Distance:  5.00 nm

--------------------------------
Race 1 was 64.34 nm long with 6 tack legs
Estimated Race Duration is 11.47 hours with 0.50 hours of Tack Penalty
============================
```

The Nineteenth Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

Problem E

Stamps

Input File: stamps.in

Philatelists have collected stamps since long before postal workers were disgruntled. An excess of stamps may be bad news to a country's postal service, but good news to those that collect the excess stamps. The postal service works to minimize the number of stamps needed to provide seamless postage coverage. To this end you have been asked to write a program to assist the postal service.

Envelope size restricts the number of stamps that can be used on one envelope. For example, if 1 cent and 3 cent stamps are available and an envelope can accommodate 5 stamps, all postage from 1 to 13 cents can be "covered":

Postage	Number of 1¢ Stamps	Number of 3¢ Stamps
1	1	0
2	2	0
3	0	1
4	1	1
5	2	1
6	0	2
7	1	2
8	2	2
9	0	3
10	1	3
11	2	3
12	0	4
13	1	4

Although five 3 cent stamps yields an envelope with 15 cents postage, it is not possible to cover an envelope with 14 cents of stamps using at most five 1 and 3 cent stamps. Since the postal service wants maximal coverage without gaps, the maximal coverage is 13 cents.

Input

The first line of each data set contains the integer *S*, representing the maximum of stamps that an envelope can accommodate. The second line contains the integer *N*, representing the number of sets of stamp denominations in the data set. Each of the next *N* lines contains a set of stamp denominations. The first integer on each line is the number of denominations in the set, followed by a list of stamp denominations, in order from smallest to largest, with each denomination separated from the others by one or more spaces. There will be at most *S* denominations on each of the *N* lines. The maximum value of *S* is 10, the largest stamp denomination is 100, the maximum value of *N* is 10.

The input is terminated by a data set beginning with zero (S is zero).

Output

Output one line for each data set giving the maximal no-gap coverage followed by the stamp denominations that yield that coverage in the following format:

```
max coverage = <value> : <denominations>
```

If more than one set of denominations in a set yields the same maximal no-gap coverage, the set with the fewest number of denominations should be printed (this saves on stamp printing costs). If two sets with the same number of denominations yield the same maximal no-gap coverage, then the set with the lower maximum stamp denomination should be printed. For example, if five stamps fit on an envelope, then stamp sets of 1, 4, 12, 21 and 1, 5, 12, 28 both yield maximal no-gap coverage of 71 cents. The first set would be printed because both sets have the same number of denominations but the first set's largest denomination (21) is lower than that of the second set (28). If multiple sets in a sequence yield the same maximal no-gap coverage, have the same number of denominations, and have equal largest denominations, then any one of the sets is acceptable.

Sample Input

```
5
2
4 1 4 12 21
4 1 5 12 28
10
2
5 1 7 16 31 88
5 1 15 52 67 99
6
2
3 1 5 8
4 1 5 7 8
0
```

Output for the Sample Input

```
max coverage =  71 :  1  4 12 21
max coverage = 409 :  1  7 16 31 88
max coverage =  48 :  1  5  7  8
```

The Nineteenth Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

Problem F

Theseus and the Minotaur

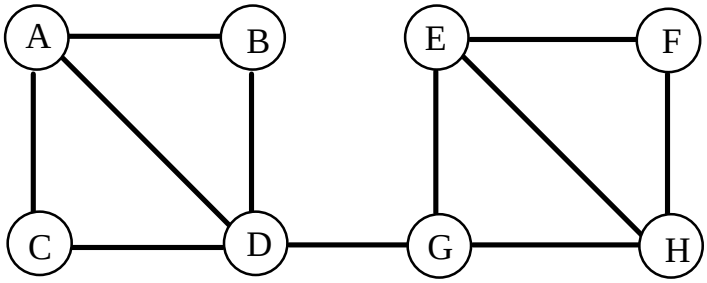
Input File: theseus.in

Those of you with a classical education may remember the legend of Theseus and the Minotaur. This is an unlikely tale involving a bull-headed monster, lovelorn damsels, balls of silk and an underground maze full of twisty little passages all alike. In line with the educational nature of this contest, we will now reveal the true story.

The maze was a series of caverns connected by passages. Theseus managed to smuggle into the labyrinth with him a supply of candles and a small tube of phosphorescent paint with which he could mark his way, or, more specifically, the exits he used. He knew that he would be lowered into a passage between two caverns, and that if he could find and kill the Minotaur he would be set free. His intended strategy was to move cautiously along a passage until he came to a cavern and then turn right (he was left-handed and wished to keep his sword away from the wall) and feel his way around the edge of the cavern until he came to an exit. If this was unmarked, he would mark it and enter it; if it was marked he would ignore it and continue around the cavern. If he heard the Minotaur in a cavern with him, he would light a candle and kill the Minotaur, since the Minotaur would be blinded by the light. If, however, he met the Minotaur in a passage he would be in trouble, since the size of the passage would restrict his movements and he would be unable to either light a candle or fight adequately. When he entered a cavern that had been previously entered by the Minotaur he would light a candle and leave it there and then turn right (as usual) but take the Minotaur's exit.

In the meantime, the Minotaur was also searching for Theseus. He was bigger and slower-moving but he knew the caverns well and hence, unlikely as it may seem, every time he emerged from a passage into a cavern, so did Theseus, albeit usually in a different one. The Minotaur turned left when he entered a cavern and traveled clockwise around it until he came to an unmarked (by him) exit, at which point he would mark it and take it. If he sensed that the cavern he was about to enter had a candle burning in it, he would turn and flee back up the passage he had just used, arriving back at the previous cavern to complete his 'turn.'

Consider the following labyrinth as an example



Assume that Theseus starts off between A and C going toward C, and that the Minotaur starts off between F and H going toward H. After entering C, Theseus will move to D, whereas the Minotaur, after entering H will move to G.

Theseus will then move towards G while the Minotaur will head for D and Theseus will be killed in the corridor between D and G. If, however, Theseus starts off as before and the Minotaur starts off between D and G then, while Theseus moves from C to D to G, the Minotaur moves from G to E to F. When Theseus enters G he detects that the Minotaur has been there before him and heads for E, and not for H, reaching it as the Minotaur reaches H. The Minotaur is thwarted in his attempt to get to G and turns back, arriving in H just as Theseus, still 'following' the Minotaur arrives in F. The Minotaur tries E and is again thwarted and arrives back at H just as Theseus arrives in hot pursuit. Thus the Minotaur is slain in H.

Write a program that will simulate Theseus' pursuit of the Minotaur.

Input

Input will consist of a series of labyrinths. Each labyrinth will contain a series of cavern descriptors, one per line. Each line will contain a cavern identifier (a single upper case character) followed by a colon (`:`) and a list of caverns reachable from it (in counterclockwise order). No cavern will be connected to itself. The cavern descriptors will not be ordered in any way. The description of a labyrinth will be terminated by a line starting with a `@` character, followed by two pairs of cavern identifiers. The first pair indicates the passage in which Theseus starts, and the second in which the Minotaur starts. The travel in a starting passage is toward the cavern whose identifier is the second character in the pair. The file will be terminated by a line consisting of a single #.

A final encounter is possible for each input data set.

Output

Output will consist of one line for each labyrinth. Each line will specify who gets killed and where. Note that if the final encounter takes place in a passage it should be specified from Theseus' point of view. Follow the format shown in the example below exactly, which describes the situations referred to above.

Sample Input

```
A:BCD
D:BACG
F:HE
G:HED
B:AD
E:FGH
H:FEG
C:AD
@ACFH
A:BCD
D:BACG
F:HE
G:HED
B:AD
E:FGH
H:FEG
C:AD
@ACDG
#
```

Output for the Sample Input

```
Theseus is killed between D and G
The Minotaur is slain in H
```

The Nineteenth Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

Problem G

Train Time

Input File: train.in

City transportation planners are developing a light rail transit system to carry commuters between the suburbs and the downtown area. Part of their task includes scheduling trains on different routes between the outermost stations and the metro center hub.

Part of the planning process consists of a simple simulation of train travel. A simulation consists of a series of scenarios in which two trains, one starting at the metro center and one starting at the outermost station of the same route, travel toward each other along the route. The transportation planners want to find out where and when the two trains meet. You are to write a program to determine those results.

This model of train travel is necessarily simplified. All scenarios are based on the following assumptions.

1. All trains spend a fixed amount of time at each station.
2. All trains accelerate and decelerate at the same constant rate. All trains have the same maximum possible velocity.
3. When a train leaves a station, it accelerates (at a constant rate) until it reaches its maximum velocity. It remains at that maximum velocity until it begins to decelerate (at the same constant rate) as it approaches the next station. Trains leave stations with an initial velocity of zero (0.0) and they arrive at stations with terminal velocity zero. Adjacent stations on each route are far enough apart to allow a train to accelerate to its maximum velocity before beginning to decelerate.
4. Both trains in each scenario make their initial departure at the same time.
5. There are at most 30 stations along any route.

Input

All input values are real numbers. Data for each scenario are in the following format.

d_1 d_2 ... d_n 0.0	For a single route, the list of distances (in miles—there are 5,280 feet in a mile) from each station to the metro center hub, separated by one or more spaces. Stations are listed in ascending order, starting with the station closest to the metro center hub (station 1) and continuing to the outermost station. All distances are greater than zero. The list is terminated by the sentinel value 0.0.
v	The maximum train velocity, in feet/minute.
s	The constant train acceleration rate in feet/minute^2.
m	The number of minutes a train stays in a station.

The series of runs is terminated by a data set which begins with the number –1.0.

Output

For each scenario, output consists of the following labeled data.

1. The number of the scenario (numbered consecutively, starting with scenario #1).
2. The time when the two trains meet in terms of minutes from starting time. All times must be displayed to one decimal place. Also, if the trains meet in a station, output the station number where they meet.
3. The distance in miles between the metro center hub and the place where the two trains meet. Distances must be displayed to three decimal places.

Sample Input

```
15.0 0.0
5280.0
10560.0
5.0
3.5 7.0 0.0
5280.0
10560.0
2.0
3.4 7.0 0.0
5280.0
10560.0
2.0
-1.0
```

Output for the Sample Input

```
Scenario #1:
   Meeting time: 7.8 minutes
   Meeting distance: 7.500 miles from metro center hub

Scenario #2:
   Meeting time: 4.0 minutes
   Meeting distance: 3.500 miles from metro center hub, in station 1

Scenario #3:
   Meeting time: 4.1 minutes
   Meeting distance: 3.400 miles from metro center hub, in station 1
```

The Nineteenth Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

Problem H

Uncompress

Input File: uncomp.in

A simple scheme for creating a compressed version of a text file can be used for files which contain no digit characters. The compression scheme requires making a list of the words in the uncompressed file. When a non-alphabetic character is encountered in the uncompressed file, it is copied directly into the compressed file. When a word is encountered in the uncompressed file, it is copied directly into the compressed file only if this is the first occurrence of the word. In that case, the word is put at the front of the list. If it is not the first occurrence, the word is not copied to the compressed file. Instead, its position in the list is copied into the compressed file and the word is moved to the front of the list. The numbering of list positions begins at 1.

Write a program that takes a compressed file as input and generates a reproduction of the original uncompressed file as output. You can assume that no word contains more than 50 characters and that the original uncompressed file contains no digit characters.

For the purposes of this problem, a word is defined to be a maximal sequence of upper- and lower-case letters. Words are case-sensitive—the word `abc` is not the same as the word `Abc`. For example,

`x-ray`	contains 2 words:	`x` and `ray`
`Mary's`	contains 2 words:	`Mary` and `s`
`It's a winner`	contains 4 words:	`It` and `s` and `a` and `winner`

There is no upper limit on the number of different words in the input file. The end of the input file is signified by the number 0 on a line by itself. The terminating 0 merely indicates the end of the input and should not be part of the output produced by your program.

Sample Input

```
Dear Sally,

   Please, please do it--1 would 4
Mary very, 1 much.  And 4 6
8 everything in 5's power to make
14 pay off for you.

   -- Thank 2 18 18--
0
```

Output for the Sample Input

```
Dear Sally,

   Please, please do it--it would please
Mary very, very much.  And Mary would
do everything in Mary's power to make
```

it pay off for you.

-- Thank you very much--